A METHOD FOR CONTROLLING THE ORDER OF DATAGRAMS

TECHNICAL FIELD

The present invention relates to a method for controlling the order of datagrams processed in a processing system of multiple processors or multi-tasking operating systems or the like.

BACKGROUND OF THE INVENTION

Multiple processors comprise an array of processing elements that contain arithmetic and logic processing circuits organized as a set of data paths. To achieve high performance the processing elements are arranged to perform tasks in parallel. These may be MIMD-based network processors or SIMD-based network processors etc. In such computer systems, which allow several processes to co-exist (e.g. multi-tasking operating system or multi-processor systems), a means to synchronize the processes is needed.

In conventional network processor, data packets or datagrams are given to processing elements on demand, as the processing elements become available. As the processing time per packet varies, the processing elements will generally not finish their work in packet order. To preserve the packet order at the output port, a random access packet queue is required. Processing elements keep track of where in the queue their packet came from and write back to the same location. The problem with the random access packet queue is its complexity. Such complexity makes it difficult to operate at the high packet rates that modern networks must sustain. For example, at OC-768 speed, the queue must handle about 100 million packets per second.

One solution is a "deli counter" algorithm. This permits the processing elements to process data packets in the order that the requests arrived. This algorithm is based on a "supermarket deli counter" in which the customer takes a ticket from a dispenser and waits until their ticket is called. The customer is then served and gives up their ticket. A counter maintains the ticket number and when a server becomes free, the counter is incremented and the next waiting customer is served. In this way the customers are served in the order that they took a ticket. In computer systems, the algorithm enables data packets to be processed in order. If a processing element is not available, the data packet retrieves a ticket from a ticket dispenser and waits until its ticket

number is called by a processing element which has become free. The ticket number is maintained and incremented by a counter whenever a processing element becomes available and the processing element accepts the next waiting data packet which is holding the corresponding ticket number. Once the data packet is retrieved, the ticket is "given up". Once the ticket counter reaches its maximum, the ticket numbers are reused. The maximum of the ticket counter would have to be sufficient to avoid duplication of ticket numbers being used by waiting data packets.

Since the processing elements may process data packets at different rates, the order of the data packets output from the processing engines cannot be preserved with conventional deli counter algorithms. Therefore, conventional deli counter algorithms still require means of preserving packet order at the output of the processing engine.

SUMMARY OF THE INVENTION

The object of the present invention is to provide a method which is capable of allocating datagrams (units of work) or groups of datagrams to processing elements that preserves global data packet order of the work units.

The method is applicable to any data flow processing system that contains multiple processors of the same or different types, for which the order of data elements must be preserved. In particular, the method is applicable to network processors that contain multiple processing elements that are served from a common source.

According to a first aspect of the present invention, there is provided a method for controlling the order of datagrams, the datagrams being processed by at least one processing engine, each of the at least one processing engine having at least one input port and at least one output port, wherein each datagram or each group of datagrams has a ticket associated therewith, the ticket being used to control the order of the datagram or group of datagrams at the at least one input port of the processing engine and at the at least one output port of the processing engine.

The processing engine may comprise a single or a plurality of processing elements. Some of the input ports and output ports of the processing elements share an input and output of the processing engine and hence a ticket.

According to a second aspect of the present invention, there is provided a processing engine for processing datagrams in a predetermined order, the processing engine comprising at least one input port, at least one output port and at least one processing element, the at least one processing element comprising an input port connected to the at least one input port of the processing engine, an output port connected to the at least one output port of the processing engine and arithmetic and logic means, the order of processing datagrams being controlled at the at least one input port of the processing engine and the at least one output port of the processing engine by a ticket associated with the datagram or a group of the datagrams.

According to a third aspect of the present invention, there is provided a processing system comprising a plurality of processing engines for processing datagrams in a predetermined order, the processing engine comprising at least one input port, at least one output port and at least one processing element, the at least one processing element comprising an input port connected to the at least one input port of the processing engine, an output port connected to the at least one output port of the processing engine and arithmetic and logic means, the order of processing datagrams being controlled at the at least one input port of the processing engine and the at least one output port of the processing engine by a ticket associated with the datagram or a group of the datagrams.

The context of which this invention is particularly applicable is a data flow processing system that processes units of data, referred to as datagrams, and that contains multiple independent processors. A datagram represents any kind of a unit of data that requires some processing. A processor in this context is any kind of programmable or non-programmable mechanism that does some transformation of the data unit, possibly but not necessarily reading or modifying some global variables as a side effect. A processor could be a programmable CPU or a fixed-function application specific integrated circuit ASIC. If the physical processor supports multiple threads, the processor can be virtual, i.e. one of several threads running on a physical CPU.

The method according to the present invention offers great flexibility in that the processors can remove or inject work units at will, and processors can join or leave the process.

BRIEF DESCRIPTION OF DRAWINGS

Figure 1 is a schematic block diagram illustrating a simplified system utilising the method according to a preferred embodiment of the present invention;

Figure 2 is a schematic block diagram illustrating a typical multiple processor system utilising the method according to a preferred embodiment of the present invention;

Figure 3 is a simplified block diagram illustrating the global semaphore unit of figure 2 according to the preferred embodiment of the present invention;

Figure 4 illustrates the ticket semaphore initialised with a queue for data according to the preferred embodiment of the present invention;

Figure 5 illustrates the Input Buffer semaphores according to the preferred embodiment of the present invention;

Figure 6 illustrates the Output Buffer semaphores according to a preferred embodiment of the present invention;

Figure 7 illustrates the processing elements on start-up according to the preferred embodiment of the present invention;

Figure 8a illustrates Ticket semaphore and Data FIFO after MTAP processor given Tickets according to the preferred embodiment of the present invention; and

Figure 8b illustrates Ticket semaphore and Data FIFO after a processing element has returned its Ticket according to the preferred embodiment of the present invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Figure 1 shows a simplified system utilising the method (or algorithm) according to a preferred embodiment of the present invention. The multiple processing system 100 comprises a plurality of processing elements 110a, 110b and 110c. The processing elements 110a, 110b, 110c may be substantially similar, or even run at the same speed; indeed one might be a programmable CPU and another an ASIC running a fixed algorithm. Datagrams, that is, the

natural unit of data in an application, (in a Network processor this would equivalent to a data packet) are supplied to a processing engine from a data source 120. The data source 120 is a functional unit that supplies one or more datagrams from a processor when requested by that processor. Although a single data source is shown here, it can be appreciated that the system may comprise a plurality of data sources. The datagrams are processed by a processing element 110a, 110b, 110c. Upon completion of the processing, the processing element 110a, 110b, 110c writes the datagram to a data sink 130 in the same order that the datagrams were read from the data source120. The data sink 130 is a functional unit that accepts processed datagrams from a processing element when requested by that processing element. Although a single data sink is shown here, it can be appreciated that the system may comprise a plurality of data sinks. The processing elements 110a, 110b, 110c can drop selected datagrams (i.e. not send them to the data sink) and can inject new datagrams (i.e. create and send a new datagram to the data sink); and processors can enter and leave the processing sequence at any time. The overall state of the datagrams needed by the deli counter algorithm is maintained by another functional unit, the semaphore unit 140.

Figure 2 illustrates an example of a parallel processing system 200 incorporating the deli counter algorithm according the preferred embodiment of the present invention. The system 200 comprises a Network Input Port (NIP) 202 and Network Output Port (NOP) 204 and a plurality of multi threaded array processors (MTAPs) 206 connected to a common bus 208. The MTAP is a single instruction multiple data (SIMD) processor that shares instruction fetch and decode amongst a number of processing elements. Typically the processing elements all execute the same instruction in lock step. In the preferred embodiment of the present invention the system includes 4 MTAPs 206a, 206b, 206c and 206d (not specifically shown in figure 2). Each MTAP 206a, 206b, 206c, 206d has, typically, at least 64 processing elements.

A Global Semaphore Unit 210 is connected to the common bus 208 to synchronise several processes over the plurality of MTAPs 206a, 206b, 206c, 206d. Each MTAP 206 is able to execute its own core instruction stream independently of the other MTAPs. The MTAPs 206 are responsible for managing the packet flow through the system. All communication between the MTAPs 206 is via the other functional blocks including external memory

(not shown in Figure 2), the Global Semaphore Unit 210, NIP 202, NOP 204 etc using the common bus 208 such that the MTAPs 206 communicate with each other by means of semaphores.

The function of the system is defined by the software running on the MTAPs 206a, 206b, 206c, 206d. Although in the preferred embodiment a plurality of substantially similar MTAPs 206 are described, the present invention can be utilised in any multi processor system in which the processor blocks may be of different types.

The massively parallel processor block at the heart of the architecture is used in a similar way to conventional processors, in that it reads and writes data in response to executing a stored program.

The semaphore unit 210 has a number of features. One feature is the ability to maintain a set of semaphores. Each semaphore has associated with it some data which is returned to the client that performs the **wait;** in the method of the present invention these are called *Tickets*. The number of Tickets is equal to or greater than the total number of buffers. In the case if the processor illustrated in figure 2 there are four buffers per MTAP 206a, 206b, 206c, 206d and four MTAPs making a total of sixteen buffers. Therefore, the number of Tickets would be greater than or equal to sixteen. Having a ticket gives the holder permission to continue but is not a handle to a particular buffer itself.

On any one MTAP 206a, 206b, 206c, 206d there are at least three threads, one requesting input of data, one for compute and one requesting output. Each thread makes use of a number of semaphores. Some of the semaphores are *local*, that is local to a MTAP 206a, 206b, 206c, 206d and some are *global* which may reside either in the global semaphore unit 210 or in specific blocks such as distributers and collectors.

The global semaphore unit 210, as shown in figure 3 comprises a ticket dispenser 302. The ticket dispenser 302 includes a FIFO buffer 304 for storing the tickets (semaphores with data) and a counter 306. The global semaphore unit 210 also comprises means 308 for generating and maintaining an Input Buffer Semaphore array and means 310 for generating and maintaining an

Output Buffer Semaphore array.

The global semaphore unit 210 is used by the software executing on the MTAPs 206a, 206b, 206c, 206d as a generic control mechanism. The global semaphore unit 210 maintains semaphores on which clients can wait and signal. It also maintains semaphores that have items of data attached. This data is returned to the client that successfully waits on the particular semaphore.

The method is based on cooperating sequential processes using shared global semaphores for synchronisation. Within the semaphore unit there is a semaphore-with-data. A semaphore-with-data comprises a semaphore and a FIFO queue. A semaphore is a counter with the atomic operations signal and wait. Sequential processes work together using semaphores as follows. Initially, a semaphore has a non-negative integer count. When a process performs a wait operation (waits on) a semaphore, the semaphore count is decremented by one, and if the resulting count is negative, the process is blocked from further execution. When a process performs a signal operation (signals) a semaphore, the semaphore count is incremented by one, and if the resulting count is non-positive, a process that is currently blocked on the semaphore is permitted to continue execution.

A global semaphore unit 210 is attached to a bus 208 which can be used by the software to synchronise behaviour. Each semaphore in the unit is memory mapped into the address space of the semaphore unit 210. The number of semaphores provided by a unit 210, and the number of units attached to the bus 208 can be tuned to the application.

To implement the above, each semaphore would have to count each signal received when there are not a currently pending wait and also queue a list of pending waits if no signals were available to satisfy them. An extension of this is to allow a small item of data to be attached to each signal which is returned to the wait that is matched with it. This requires signals to be queued and not just counted.

All semaphore state is accessible via the bus, to allow the unit's context to be saved/restored or otherwise modified.

A semaphore-with-data has the following additional behaviour. When a semaphore-with-data is signalled, the

signalling process provides a value as a calling argument. The semaphore associated with the semaphore-with-data is dealt with as described above, and in addition a copy of the argument value is inserted into the associated FIFO. When a semaphore-with-data is waited on, a wait operation is performed on the associated semaphore as described above. In addition, a value extracted from the associated FIFO and returned to the calling process.

The method according to the preferred embodiment comprises a system initialisation section and an operation section.

In the initialisation section, in which up to N processors can participate in the method. Figure 3 shows the contents of the global semaphore unit after initialisation, with N=8.

The semaphore-with-data (ticket dispenser 302), in the global semaphore unit 210 is initialised as follows: the semaphore part is initialised with the value N; and the FIFO part is filled with a sequential set of N values, 0 to N-1, called tickets.

An array 308 of N semaphores, called InBuf, in the global semaphore unit are allocated for purposes of reading from the data source. The k-th element of this array is referred to as InBuf[k]. These semaphores are initialised to have a count of zero except for the first (InBuf[0]) which is initialised to have a count of one.

An array 310 of N semaphores, called OutBuf, in the global semaphore unit are allocated for purposes of writing to the data sink. The k-th element of this array is referred to as OutBuf[k]. These semaphores are initialised to have a count of zero except for the first 9OutBuf[0]) which initialised to have a count of one.

The operation section specifies the behaviour of one of the processors that is participating in the method, i.e. reading datagrams from the data source, transforming them, and writing them to the data sink.

1. Take a ticket, that is wait on TicketDispenser, obtaining a ticket (T).
2. Return the ticket, that is signal TicketDispenser, with value T.
3. Wait on InBuf[T].
4. Read a datagram from the data source.
5. Signal InBuf[T+1 mod N].

6. Transform the datagram.
7. Wait on OutBuf[T].
8. Write the datagram to the data sink.
9. Signal OutBuf[T+1 mod N]
10. Go to step 1.

The number N of tickets is determined by the number of processors that can be concurrently processing datagrams. The only requirement is that N is at least as large as the number of processors. The method preserves order of the datagrams. That is, the datagrams are delivered to the output sink in the order they were taken from the input source. To see this, note that initially exactly one of the InBuf elements has the valuel. This means that all the processors will block at step 3 except for the processor that has ticket 0. That processor will read a datagram in step 4 and set InBuf[1] to 1 in step 5. This permits the processor holding ticket 1 to proceed from step 3. In general after the algorithm has run for a while, at most one of the InBuf semaphores will have the value 1 and the rest will have the value 0. If all the InBuf semaphores are currently 0, one of the processors will eventually execute step 5, causing an InBuf semaphore to be set to 1. This effectively permits the processor with the ticket value corresponding to that InBuf semaphore to proceed to read the next datagram. On the output side, a similar argument applies. At most one of the OutBuf semaphores will have the value 1 and the rest will have the value 0. If all the OutBuf semaphores are currently 0, one of the processors will eventually execute step 9, causing an OutBuf semaphore to be set to 1. This permits the processor with the ticket value corresponding to that OutBuf semaphore to proceed to write the next datagram.

Which of the processors actually handle any datagram is irrelevant. Whichever processor it is will notify the processor that should read next by executing step 5 and will notify the processor that should write next by executing step 9. When a processor takes a ticket, it is committing itself to execute 3, 5, 7 and 9 to keep the sequence going properly.

A new processor can join the algorithm at any time by simply taking a ticket and following the basic steps given above. A processor can drop out of the algorithm at any time by simply leaving the flow above at step 9.

In order to stop a datagram, a processor simply omits to execute the write step 8. To inject a datagram, a processor simply omits to execute the read step 4.

The method according to the preferred embodiment can handle multiple data sinks. The method is extended to handle multiple data sources by assigning a ticket dispenser semaphore-with-data and a pair of arrays of semaphores to each data source. Then steps 1 and 2 of the method are modified as follows:

1. Select a TicketDispenser. Take a ticket T from that TicketDispenser. That is, wait on the selected TicketDispenser, returning a ticket value T.
2. Return the ticket. That is, signal with TicketDispenser selected in step 1 with value T.

The remaining steps are the same, except that they must use semaphores and data source associated with the select ticket dispenser. The method does not specify which of the ticket dispenser should be selected in step 1. The choice could be arbitrary, or the most full ticket dispenser could be selected. A possible variation is that different processors may choose from different sets of ticket dispensers, e.g. if different data sources should be serviced by different processors.

In the mutiple processing system shown in figure 2, the semaphores used are:

- Local semaphores:

  FreeBuffer, is initialised to four

  FullBuffer, is initialised to zero

  ComputeBuffer, is initialised to zero

- Global semaphores:

  InputBufferSemaphore(16), this is an array of semaphores,

  InputBufferSemaphore(0) is initialised to one and **InputBufferSemaphore(1..15)** is initialised to zero.

  NIPRequestSemaphore, initialised to four.

  OutputBufferSemaphore(16), this is an array of semaphores,

  OutputBufferSemaphore(0) is initialised to 1 and **OutputBufferSemaphore(1..15)** is initialised to zero.

  CollectorSemaphore, initialised to one.

Examples of Input Buffer semaphores and Output Buffer semaphores are shown in figures 5 and 6 respectively.

There is also a FIFO of Tickets, which is used to communicate Ticket numbers from the Input thread to the Output thread.

Input Thread

```
    while (true) {
      wait FreeBuffer                    // Local semaphore
      GetTicket (K)                      // Get Ticket value
      PutLocalTicket (K)                 // Place in local
    FIFO for output thread
      wait InputBufferSemaphore (K) // Wait on ticket
    semaphore
      wait NipRequestSemaphore        // wait for NIP (May
    not be necessary, e.g. have a queue in the
    Distributor that does not overflow)
      ReadNIP                            // Issue request to
    NIP
    signal NipRequestSemaphore     // Release NIP (May not
    be necessary, e.g. have a queue in the Distributor
    May not be necessary, e.g. have a queue in the
    Distributor that does not overflow)

      PutTicket (K)                      // Put back ticket
    value
      //
      // Signal the next ticket semaphore
      //
      signal InputBufferSemaphore
    ((K+1)%TotalNumberOfBuffer)
      wait ReadComplete                  // Hold off until DIO
    finished
      signal FullBuffer                  // Buffer available
    for compute thread
    }
```

Compute Thread

For illustration this is doing both lookups and regular compute

```
    while (true) {
        wait FullBuffer        // Wait until buffer
    available for compute
      ComputeOperation
```

```
  signal ComputeBuffer // Indicate buffer finished
with and can be emptied
}
```

Output Thread

```
while (true) {
  wait ComputeBuffer                   // Wait until a
buffer is available
  GetLocalTicket (K)                   // Get Value used
by Input thread
  wait OutputBufferSemaphore (K)       // wait for correct
output turn
  wait CollectorSemaphore              // wait for
Collector (This is actually implemented in the LUE
transfer engine)
  write NOP                            // Issue request to
NOP
  //
  // Indicate okay for next in turn
  //
  signal OutputBufferSemaphore
((K+1)%TotalNumberOfBuffer)
      wait WriteComplete                   // wait for DIO
to complete
  signal CollectorSemaphore            // Release
collector (This is actually implemented in the LUE
transfer engine)
  signal FreeBuffer                    // Return the
buffer
}
```

On start-up each MTAP will get a ticket (wait on a semaphore with data) using **GetTicket**. The MTAP whose request reaches the semaphore block first will be given ticket value 0 and since the associated **InputBufferSemaphore** has been pre-signalled it will enable the execution of the Input thread. All the other processors will **wait**.

As shown in figure 4, for a system where the MTAP are serviced in a round robin manner, the data 400(0) through to 400(15) in the FIFO of Tickets attached to the Ticket semaphore is initialised in numerically increasing values. The semaphore (or count) 410 is pre-initialised

to the number of items in the data FIFO - in this case 16.

Figure 7 shows an example where the MTAPs 206a, 206b, 206c, 206d have started up and the requests to the semaphore block have resulted in the order A, C, B, D, for example, i.e. MTAP 206a is given ticket 0, MTAP 206c is given ticket 1, MTAP 206b is given ticket 2 and MTAP 206d is given ticket 3. This will be the order of round robin sequence.The state of the Ticket semaphore is shown in figure 8a .

Once the Input thread of processor 206a has issued its read request to the NIP 202 it will put back its current ticket and signal the next **InputBufferSemaphore.** The ticket value sent back to the Ticket semaphore will be placed at the end of the linked list as shown in Figure 8b and the associated count will be incremented. This will allow MTAP processor 206c to proceed with it's Input thread, which was **waiting** on the **InputBufferSemaphore** associated with its ticket value.

MTAP 206a's Input thread will now **wait** on it's local semaphore **ReadComplete**, which will be signalled by the Direct I/O (DIO) mechanism for that processor. Once the DIO has completed its operation the Input thread will **signal**, a local semaphore **FullBuffer**, on which the Compute thread is **waiting**. The Input thread is now ready to start its set of operations all over again.

MTAP 206a's Compute thread can now proceed with computation and lookups - for illustration the lookup and compute is being performed by one thread, however in a realistic system there will more than one thread doing this. Once the compute has completed a local semaphore, **ComputeBuffer**, is signalled.

MTAP 206a's Output thread can now proceed as it has been **waiting** on the local semaphore **ComputeBuffer**, which has been signalled by the Compute thread. The Output thread now **wait** on a global semaphore in **OutputBufferSemaphore**. This is actually an array of semaphores, which at start-up will be initialised to the same semaphore values as the **InputBufferSemaphore.** That is initially only the first element of the array will be pre-signalled. The Output processor will issue a request to the Collector and **signal** the next global semaphore in the array **OutputBufferSemaphore.**

Should the need arise where a MTAP needs to drop out of the round robin sequence, then this can be achieved by taking a ticket, **signalling** the next **InputBufferSemaphore** immediately and missing out the NOP phase. The thread sequencing will then look like this.

Input Thread

```
while (true) {
        wait FreeBuffer                          // Local
semaphore
   GetTicket (K)                             // Get Ticket value
   PutLocalTicket (K)                        // Place in local
FIFO for output thread
   wait InputBufferSemaphore (K) // Wait on ticket
semaphore
        PutTicket (K)                            // Put back
ticket value

   signal InputBufferSemaphore
((K+1)%TotalNumberOfBuffer)
   signal ComputeBuffer                      // compute thread
is skipped
}
```

Compute Thread

The compute thread is not used.

Output Thread

```
while (true) {
   wait ComputeBuffer                        // Wait until a
buffer is available to empty
   GetLocalTicket (K)                        // Get Value used
by Input thread
   wait OutputBufferSemaphore (K)  // wait for correct
output turn

   signal OutputBufferSemaphore
((K+1)%TotalNumberOfBuffer)
   signal FreeBuffer                         // Return the
buffer
}
```

As mentioned previously another processor can join the sequence by taking a ticket and skipping the NIP phase. That will require more InputBufferSempahores and

OutputBufferSemaphores. It is up to the user how many more are needed. In case the new processor only needing occasional access, the dropping out of sequence method can be used.

In it's simplest version, each semaphore occupies Xbytes of the address space. Each write to that address is recognised as a signal. Each read is recognised as a wait. The read does not return data until a signal has been received. The value written is discarded and the value returned is undefined. The maximum number of pending signals or waits is one. i.e. there is no counting or queuing. This can be extended to allow the value contained in the signalling write to be returned in the waiting read. It can also be extended to allow multiple pending signals and waits. This requires the signals be counted and waits be queued. We may choose that the values written are ignored, or choose that the be added to the semaphore count value. We may choose that the values read are undefined or that the contain the current value of the semaphore counter. This can be further extended to allow the value contained in the signalling write to be returned in the read of the wait to which it is matched. This replaces the counter increment/examine options. It also requires that a queue is maintained of the pending signals instead of just counting. In one alternative arrangement the NIP/NOP would operate such that they decide which mini-cores receive packets next. However this requires that the distribution algorithm is chosen and fixed in hardware. An alternative arrangment is instedd of hardwired flow of control use software based. The objective is to allow the mini-cores/processors to decide what the algorithm is, and thus cast it in software rather than hardware. The motivation for this is two fold. Firstly it introduces flexability and secondly it simplifies the hardware.

The system bus 208 is preferrably a split transaction type, i.e. that reads are two seperate transactions, a request and a response. This would prevents deadlocks occurring. Since a deadlock would occur if a wait were issued unless a matching signal was already posted. The wait (read) would tie up the bus and prevent any signal being sent (a write) that would complete the transaction.

Although a preferred embodiment of the method and system of the present invention has been illustrated in the

accompanying drawings and described in the forgoing detailed description, it will be understood that the invention is not limited to the embodiment disclosed, but is capable of numerous variations, modifications without departing from the scope of the invention as set out in the following claims.